ConcurDB: Concurrent Query Authentication for Outsourced Databases

Sumeet Bajaj, Anrin Chakraborti, Radu Sion

Abstract—Clients of outsourced databases need *Query Authentication* (QA) guaranteeing the integrity and authenticity of query results returned by potentially compromised providers. Prior work provides QA assurances for a limited class of queries by deploying several software-based cryptographic constructs. The constructs are often designed assuming read-only or infrequently updated databases. For dynamic datasets, the data owner is required to perform all updates on behalf of clients. Hence, for concurrent updates by multiple clients, such as for OLTP workloads, existing QA solutions are inefficient. We present ConcurDB, a concurrent QA scheme that enables simultaneous updates by multiple clients. To realize concurrent QA, we have designed several new mechanisms. Firstly, we identify and use an important relationship between QA and memory checking to decouple query execution and verification. We allow clients to execute transactions concurrently and perform verifications in parallel using an offline memory checking based protocol. Then, to extend QA to a multi-client scenario, we design new protocols that enable clients to securely exchange a small set of authentication data even when using the untrusted provider as a communication hub. Finally, we overcome provider-side replay attacks. Using ConcurDB, we provide and evaluate concurrent QA for the full TPC-C benchmark. For updates, ConcurDB shows a 4x performance increase over existing solutions.

Index Terms—Database Security, Query Authentication

1 Introduction

Today, all major cloud providers offer a database service as part of their overall solution. However, cloud-based services require their customers to fully trust the provider. This is often unacceptable and technology-backed security assurances are key to clients' adoption decisions. In case of actively malicious or faulty service providers, query results cannot be trusted without proper *Query Authentication* (QA) mechanisms [29].

Existing research tackles the QA problem by designing *authenticated data structures* (ADS) that are uploaded to the provider along with the outsourced data. Using ADS, a provider constructs proofs of query execution. Clients verify the proofs to check both correctness and completeness of query results. A set of ADS have been proposed for individual query types, such as range [15, 23, 25, 31], aggregation [16], spatial [11, 18], set operations [26], and key word search [10] queries.

Sumeet Bajaj is currently with Private Machines Inc. This work was done while the author was affiliated to the Department of Computer Science, Stony Brook University, NY - 11790.
Anrin Chakraborti and Radu Sion are with the Department of Computer Science, Stony Brook University, NY-11790.
This research was supported in part by NSF awards 1526707, 1526102, and the Office of Naval Research. by the US ARMY, and by gifts from Northrop Grumman Corporation, Parc/Xerox, and Microsoft Research

QA complicates update operations since changes made to the outsourced database tuples involve modifications in the related server-side ADS. In case of concurrent updates by multiple clients, existing QA solutions have two key limitations. Firstly, current ADS designs limit transaction concurrency because in order to keep the authentication information small, they require updates to a common subset of data items in an ADS resulting in bottlenecks. Secondly, since current QA solutions do not synchronize clients, they are also subject to *replay attacks* (or *fork consistency* attacks [17]). A replay attack occurs when the server hides updates of one client from other clients by using old authentication data.

Due to the limitations of concurrency and replay attack detection, existing QA solutions do not support update operations [29]; assume fairly static or infrequently updated databases [7, 20]; rely on the data owner to perform all updates on behalf of clients [21, 25, 34]; permit only periodic updates [23]; or suggest batching of update operations [15]. Moreover, solutions that explicitly support concurrent query authentication rely on expensive cryptographic operations [32, 33].

In this paper, we design and evaluate ConcurDB, a concurrent, multi-client QA scheme that eliminates bottlenecks on ADS updates and detects replay attacks efficiently. ConcurDB uses signature based query authentication without the use of expensive cryptographic primitives using offline memory checking. In the following, we highlight some of the key features and techniques used in ConcurDB.

Decoupling Queries and Authentication. Query authentication (QA) is typically classified into two categories: *offline query authentication* and *online query authentication*. In online QA, the results of a query are verified as soon as the query is executed. However, this comes at a significant cost: *the per query complexity of online QA cannot be better than* $\Omega(\log n/ \log\log n)$, *where* n *is the number of database tuples [8]*. Also, for a multi-client scenario, online QA results in a serial dependency between client queries.

ConcurDB adopts a more efficient approach by decoupling query execution from query authentication and performing offline QA. At a high level, this is done by creating *unforgeable links* between tuples in a relation that can be generated/updated only by the user. Links are read/updated as part of the queries for the corresponding tuples. QA is then modeled as a memory checking problem [3] for verifying the integrity of the

links between the tuples (detailed in Section 4).

ConcurDB uses offline memory checking to delay verification, which can now be performed periodically independent of query execution. In effect, QA in ConcurDB has $O(1)$ amortized per-query complexity.

Concurrent Signature-Based Query Authentication. Query authentication techniques can be broadly classified into two categories – *tree-based* and *signature-based*. Bajaj et al. [2] compared existing QA solutions based on published experimental results. They found that signature-based approaches perform better for range query processing. Tree-based approaches are more efficient for join processing. Therefore, knowledge of application query types can be a factor in the choice between tree or signature-based QA.

For cloud environments, where data transfer costs dominate [5], signature-based QA are preferred due to the small, constant *verification object* sizes. In fact, analysis of tree-based QA has shown *verification object* sizes to be $O(\log_b n)$, where n is the total number of database tuples and b is the B$^+$-tree branching factor [15].

In ConcurDB, we couple signature-based query authentication with offline memory checking to achieve a construction with a high degree of parallelism at low cost. One particular advantage of this approach is that we avoid authenticated data structures with expensive cryptographic operations for concurrency [14, 28].

Security Against Replay Attacks. Memory checking protects integrity of data but does not prevent replay attacks in a multi-client setting. This is because offline memory checking requires several authentication variables, which are stored client-side in a single client scenario. If outsourced in a multi-client scenario, the untrusted remote server can replay these variables to fork client views. To mitigate this, ConcurDB employs a novel *authentication set exchange protocol* which allows clients to update and exchange the authentication variables using the server as a communication hub while protecting against a replay attack.

Summary. The following summarizes our contributions:

- We model QA as offline memory checking [3] and design an efficient offline QA scheme for SQL queries. Our scheme preserves concurrency, efficiently detects replay attacks with amortized O(1) complexity, and avoids expensive crypto operations.
- We apply our offline QA scheme to database transactions involving multiple SQL queries, such as range (select), update, insert, delete, projection, and join (Section 4.3).
- We extend offline QA to enable replay attack detection in a multi-client scenario (Section 5.1). The extension involves new protocol designs that allow clients to securely exchange small, constant-sized authentication data in a fixed order while using the untrusted provider as a communication hub.
- We provide extensions to our scheme that enable precise identification of transactions for which QA requirements were violated (Section 5.4)
- Using ConcurDB, we realize efficient QA for the full TPC-C benchmark (Section 6).

2 Related Work

In this section, we briefly overview existing QA work to clearly identify concurrency-related limitations. The reader is directed to other external published surveys and comparisons of experimental results and overviews of various QA solutions [2].

At a high level, existing solutions work similarly using the following steps.

1) A data owner uploads a database and related *authenticated data structures* (ADS) to a service provider. In addition, the owner may distribute certain information to clients.
2) A client submits a query to the service provider.
3) The provider executes the query to get the desired results. Using the ADS, the provider also computes a proof, which the client uses to verify both correctness and completeness. The proof is referred to as a *verification object* (VO).
4) The service provider delivers both the query results and the VO to client.
5) Using information from the owner together with query results and VO from the service provider, the client determines whether QA assurances are met.

Properties of ADS and VO prevent the provider from compromising QA requirements without detection. Based on the data structures used existing QA solutions can be classified as *tree-based* or *signature-based*.

2.1 Tree-based QA Solutions

In tree-based approaches, the ADS is constructed as a tree. As part of range or join query execution, service provider traverses the tree and gathers query results and nodes that form the VO. Using the VO, client reconstructs the traversal path used in query execution to verify correctness and completeness.

The Merkle B-tree (MBT) [15, 22] forms the basis of tree-based approaches. All other tree-based approaches, such as EMBT [15], AIM [31], and XBT [25] are variations of MBT.

MBT is essentially a Merkle hash tree [19] applied to a B$^+$-tree. In a MBT, a hash value is computed for each B$^+$-tree node using a cryptographic hash function, such as SHA-1. The hash for a B$^+$-tree leaf node is computed as follows: The hash of each tuple contained in the leaf node if first computed. Then, the hashes of all the leaf node's tuples are concatenated. A hash of the concatenated value gives the leaf node's hash.

The hash of a non-leaf node is computed as the hash of concatenation of the hashes of the node's children. The root node's hash is signed by the data owner using a secret key. The root signature can be verified by clients using the owner's public key.

During query execution, the provider gathers a minimal set of tuple and B$^+$-tree node hashes that would

enable the client to reconstruct the root node's hash. The additional tuple and node hashes constitute the VO. Using the VO and query results, client reconstructs the root hash. The client then compares the computed root hash to the original root hash signed by the owner. If the two hashes match, QA is ensured [22].

Concurrency. For update operations, concurrency in tree-based solutions is limited due to the common set of hash values modified by all update transactions. An update of any tuple in the dataset requires recomputation of node hashes up to the root. Thus, in a multi-client scenario, each client update locks and modifies multiple node hashes significantly limiting transaction concurrency. In fact, even when transactions update independent tuples, they will necessarily have contention updating the root hash.

We note that other tree-based approaches are derived from the MBT approach. Hence, they too have a common set of hash values modified on each update limiting concurrency. Concurrency limitations of tree-based approaches, such as Embedded-Merkle B-tree (EMBT) [15] have been identified and experimentally demonstrated in prior work [23].

Replay Attacks. In tree-based approaches, there are two options to store the root hash. The first option is to distribute the root hash to clients when the database and ADS are initially uploaded to the provider. Although this approach precludes server-side replay attacks, it necessitates synchronization between clients to communicate updated root hash values.

The second option is to store the signed root hash with the provider. On updates, a new root hash is computed and resigned using a secret key not known to the provider. However, storing the signed root hash with the provider enables replay attacks. Using old hash values, the provider can transfer stale results to clients.

To avoid replay attacks, most QA solutions limit updates to the data owner [21, 25, 34]. Jain et al. [12] designed a QA solution that permits updates by clients. However, all transactions are verified by the data owner. Moreover, Jain et al. use a MBT ADS limiting server-side concurrency due to contention for hash updates.

2.2 Signature-based QA Solutions

In signature-based schemes [21], the ADS consists of a set of signatures forming a chain. For each tuple, a signature is computed on the concatenation of the tuple's hash along with the hash of the immediate predecessor (or successor) tuple[1]. For example, the signature for a tuple t_i may computed as $S(h(t_i)||h(t_{i-1}))$, where $S(\cdot)$ denotes a signature operation using the data owner's secret key, $h(\cdot)$ represents a cryptographic hash, and t_{i-1} is the predecessor of t_i when sorted on the search attribute. By including the predecessor in the signature, a chain of all tuples is formed ordered on the search attribute. The VO then consists of the signature of all tuples in the query result and the signature of two boundary tuples not in the result. Since each tuple is linked to its predecessor in an unforgeable manner, the client can verify that no tuple is either illicitly inserted or omitted from the query result.

Concurrency. Compared to tree-based approaches, signature-based solutions exhibit good concurrency for updates. In signature-based approaches, an update of a tuple requires updates to the tuple's and the neighboring tuples' signatures. Hence, two updates are in lock contention only if they update same or adjacent tuples.

Replay Attacks. In signature-based QA, the ADS consists of as many signatures as the number of database tuples making replay attack detection challenging.

If updates are limited to the data owner, client notification becomes an issue since for any update by the owner, all clients need to be notified of signatures that are no longer valid. Pang et al. [23] address the problem of client notification to a certain extent by allowing the data owner to periodically publish a concise bitmap of signatures that were updated in the current period. The solution requires clients to check freshness of each signature received in the VO. However, replay attacks can only be detected for a period p, if the client has stored a bitmap for p. Hence, client-side storage can potentially become a limiting factor.

2.3 Memory Checking For Integrity

Recent work by Arasu et al. [1] identifies memory checking as a tool for building concurrent key-value stores with integrity. Specifically, *Concerto* supports concurrent queries on a key-value store while trusted code on the server (executing in a trusted execution environment (TEE) such as Intel SGX) performs offline memory checking in background to verify integrity of results. However, the techniques presented in [1] are only applicable to key-value stores and cannot be extended trivially to support expressive SQL queries on a relational database.

Also, the reliance on server-side code and trusted hardware for verification may be limiting in several cases since it requires support from the cloud provider. It is unclear if the techniques presented in [1] can be applied to settings where there is no trusted hardware support and the clients do not communicate with each other, as assumed in this work.

3 Background

QA Model. Data is uploaded by a data owner to a relational database hosted with a remote, untrusted service provider. For QA, the data owner computes and places additional authentication data with the provider.

Clients authorized by the data owner query the outsourced database through a SQL interface exposed by the provider. The service provider is not trusted. Due

1. Variations of the discussed signature-based scheme can be constructed by including both predecessor and successor tuples in the signature chain or by computing signatures using multiple search attributes [24].

to compromise or malicious intent, the provider may violate one or more of the QA security requirements.

QA Security Requirements. Query Authentication has two requirements – correctness and completeness. This is fairly well defined in prior works [2, 21] and we refer to them for formal definition. In the following, we provide an intuition for these requirements.

- *Correctness:* This has two components –
  - Each tuple in the query result should be authentic, that is, all tuple's must originate from the original database that was uploaded to the provider's site. It must be impossible for the provider to introduce any spurious tuples in the result.
  - Each tuple in the query result must satisfy the query predicates.
- *Completeness:* For completeness, all tuples that are supposed to be part of the query result must be present in the result. Query execution should not exclude any valid tuples from the result.

Untrusted Provider. For ConcurDB, the provider is untrusted. The provider can reorder or delete existing tuples added by the client to the database. The provider can also return spurious tuples during queries.

Trusted Clients. The clients are assumed to be trusted and correctly execute the ConcurDB protocols. Further, the clients do not collude with the provider. *Importantly, clients do not interact with each other.*

3.1 Offline Memory Checking

Model. Memory checking [3] involves three entities: an untrusted memory $\mathcal{M}$, a trusted user $\mathcal{U}$, and a trusted checker $\mathcal{C}$. $\mathcal{M}$ consists of n registers. Each register has a b-bit unique address and can store a b-bit value. $\mathcal{C}$ is assumed to have a small amount of trusted memory of size $O(\log n)$. User $\mathcal{U}$ requests two types of operations – read and write. Given a register address a, read returns the b-bit value stored at a. Given a register address a and a b-bit value v, write stores v at address a.

User $\mathcal{U}$ sends a sequence of read and write operations to $\mathcal{C}$. $\mathcal{C}$ translates a user operation into multiple memory read and write operations. $\mathcal{C}$'s job is to verify that $\mathcal{M}$ operates correctly. $\mathcal{M}$ is said to operate correctly, if the value read by user from any register a is the latest value that was written to a [3]. The offline checker by Blum et al. stores a timestamp of size $O(\log n)$ with each register in $\mathcal{M}$. A memory read operation thus accepts an address a and returns a tuple (v, t), where v and t are the value and timestamp stored at a, respectively. A memory write operation accepts a triple (a, v, t) and stores the value v and timestamp t at a.

Offline Checker. Checker $\mathcal{C}$ stores two hash values $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ in its memory. Here, $\mathcal{R}$ is the set of all triples (a, v, t) read from $\mathcal{M}$ and $\mathcal{W}$ is the set of all triples (a, v, t) specified to write operations. Initially, $\mathcal{R} = \mathcal{W} = \phi$.

The hash function $\mathcal{H}$ has the following properties:

- *Incremental*: Given $\mathcal{H}(\mathcal{R})$ and a triple (a, v, t), $\mathcal{H}(\mathcal{R} \cup (a, v, t))$ can be computed efficiently.
- *Collision Resistant*: If $\mathcal{R} \neq \mathcal{W}$, then $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ with high probability.

For a user operation $\mathsf{write(a, v)}$, the checker performs the following actions.

- Reads the value v' and time t' stored at a.
- Checks that current time is greater than t'.
- Writes current time t and value v to a.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (a, v', t'))$
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (a, v, t))$

For a user operation $\mathsf{read(a)}$, the checker performs the following actions.

- Reads the value v' and time t' stored at a.
- Checks that current time is greater than t'.
- Writes current time t and value v' to a.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (a, v', t'))$
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (a, v', t))$

After a sequence of operations, $\mathcal{C}$ reads all n memory registers and updates $\mathcal{H}(\mathcal{R}) = \mathcal{H}(\mathcal{R} \cup (a_1, v_1, t_1) \cup (a_2, v_2, t_2), \ldots, \cup(a_n, v_n, t_n))$ where (a_i, v_i, t_i) is the tuple stored in the ith register. Effectively, after this step, $\mathcal{R} = \mathcal{W}$ and $\mathcal{H}(\mathcal{R}) = \mathcal{H}(\mathcal{W})$.

Blum et al. show that if the operator does not function correctly, e.g., replays old states of the triples, then during the check $\mathcal{R} \neq \mathcal{W}$ and $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ for a collision-resistant hash function $\mathcal{H}$. We refer to [3] for more details and formal proofs of correctness.

In Section 4.2, we show how ConcurDB leverages offline memory checking to design a query authentication mechanism where a (trusted) client functions as both the user, $\mathcal{U}$ and the checker, $\mathcal{C}$.

Complexity. The checker performs a constant number of memory operations for each user read or write. Thus, the final step wherein the checker reads all memory registers to update $\mathcal{H}(\mathcal{R})$ dominates the runtime cost. To achieve $O(1)$ amortized cost per user operation, the checker performs the final step only after n user operations have completed. Thus, memory correctness is verified only after a long sequence of n operations.

4 ConcurDB: Query Authentication

In this section, we model QA as the offline memory checking problem described in Section 3.1. Then, we extend the basic QA solution to support QA for database transactions involving multiple SQL queries, such as range, updates, inserts, deletes, projections, and joins (Sections 4.3 - 4.4). Finally, we present a novel protocol that extends offline QA to multiple clients with concurrency (Section 5.1).

Overview. For each relation, we first sort the tuples on a search attribute. Then, we create links between adjacent tuples. The links are similar to the signatures in signature-based QA (Section 2.2). However, since we

do not use expensive crypto operations, such as RSA signatures, we use the term links rather than signatures to differentiate from existing signature-based QA solutions.

We create links as a function of tuple attribute values[2]. As a result, the links also guarantee tuple integrity. Moreover, a SQL query can now be considered as a sequence of reads and writes on links. Therefore, memory checking can be applied to links (Section 4.2) along with checks for correctness and completeness (Section 4.4).

Since links are a function of tuple attributes, updates and inserts of database tuples correspond to link updates and insertions, respectively. By properties of memory checking, we ensure that, a link value read as a result of a SQL query is the latest value written to the database. Reuse of an old link value by the provider or fake tuples in the query result cannot go undetected.

4.1 Initialization and Hashing Tools

Data Upload and Links Creation. At initialization, the data owner uploads a relation $R = \{t_0, t_1, t_2, ..., t_n, t_{n+1}\}$ to the provider, where each t_i is a tuple, such that $t_i.a < t_{i+1}.a$ for some attribute a. t_0 and t_{n+1} are two fake boundary tuples, such that the values of $t_0.a$ and $t_{n+1}.a$ are never used for any other tuples. Additionally, the data owner uploads a set of triples $\mathcal{L} = \{(l_0, v_0, ts_0), (l_1, v_1, ts_1), ..., (l_n, v_n, ts_n)\}$. Each triple (l_i, v_i, ts_i) represents the following.

- l_i is a link between t_i and t_{i-1}.
  That is, $l_i = h(h(t_{i-1})||h(t_i)))$, where h is a cryptographic hash function, such as SHA.
- v_i takes two values. $v_i = 1$ means link l_i is valid and $v_i = 0$ indicates an invalid link.
- ts_i is the timestamp for link l_i. The timestamp is incremented on each read and write of l_i.

At initial upload time, $v_i = 1$ and $ts_i = 0$ for all triples (l_i, v_i, ts_i), $1 \leq i \leq n+1$.

Thus, the triples (l_i, v_i, ts_i) closely resemble the (a, v, t) triples in memory checking (Section 3.1).

Incremental Hashing. Similar to memory checking, the data owner records two hash values, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$. Initially, $\mathcal{R} = \phi$ and $\mathcal{W} = \mathcal{L}$.

As in memory checking, we require the hash function $\mathcal{H}$ to be incremental. That is given $\mathcal{H}(\mathcal{R})$ and a triple (l, v, t), it should be efficient to compute $\mathcal{H}(\mathcal{R} \cup (l, v, t))$. To ensure collision resistance, we use incremental hash functions based on modular arithmetic [6, 9][3].

Using incremental hashing with a large modulus p [27], owner computes $\mathcal{H}(\mathcal{W})$ as $\sum_{i=1}^{n+1} h(l_i||v_i||t_i) \ mod \ p$. For now, we assume that the owner distributes $\mathcal{H}(\mathcal{R})$, $\mathcal{H}(\mathcal{W})$, and a count $\mathcal{C} = |\mathcal{W}|$ to client. Later, in Section 5.1 we describe a new protocol wherein multiple clients can securely exchange the set $\{\mathcal{H}(\mathcal{R}), \mathcal{H}(\mathcal{W}), \mathcal{C}\}$ using the untrusted server as a communication hub.

2. To support projections, we construct a tuple hash as a Merkle hash tree (MHT) on the tuple's attribute values. The MHT root hash is the tuple's hash.

3. Proved collision-resistant via equivalence to the weighted subset sum problem.

Algorithm 1 cwrite(l)

1: Read (v, ts) for link l.
2: **if** $v \neq 1$ **then**
3:     Abort
4: **end if**
5: Write $(l, 0, ts+1)$
6: $\mathcal{H}(\mathcal{R}) = \mathcal{H}(\mathcal{R}) \cup (l, v, ts)$
7: $\mathcal{H}(\mathcal{W}) = \mathcal{H}(\mathcal{W}) \cup (l, 0, ts+1)$

Algorithm 2 cwrite_new(l)

1: Let $\mathcal{C} \leftarrow$ Query Identifier
2: Write $(l, 1, 0)$
3: $\mathcal{H}(\mathcal{W}) = \mathcal{H}(\mathcal{W}) \cup (l, 0, ts+1)$
4: $\mathcal{C} = \mathcal{C} + 1$

4.2 BasicQA

For a sequence of link reads and writes, our BasicQA protocol ensures correctness and detection of replay attacks. That is, any link returned by provider as a result of a read, is either the original link uploaded by the owner or a new link uploaded by client. If a link is made invalid by client, then the invalid link cannot be returned as a result of any future read without detection.

Provider and Client operations. In BasicQA, the provider supports two operations – $read(l)$ and $write(l, v, ts)$. Read returns the tuple (v, ts), where v is the value and ts is the timestamp, respectively, for link l. $write(l, v, ts)$ stores the triple (l, v, ts) at the provider.

A client in BasicQA functions as both the user, $\mathcal{U}$ and the checker, $\mathcal{C}$. Thus, the authentication hashes for the memory checker, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ are stored client-side. This requires $O(\log n)$ storage. This has an additional benefit – since the clients are trusted, additional security mechanisms are not required to protect the hashes.

To capture link invalidation (as part of update queries) and addition of new links (as part of insert and update queries) by client, BasicQA supports three operations on the client's side – cwrite, cwrite_new, and cread. cwrite invalidates an existing link and cwrite_new stores a new valid link with the provider.

Invalidating Links (Algorithm 1). On a cwrite(l), a client performs the following.

- Reads the tuple (v, ts) for link l from the provider.
- Checks that v is 1, that is, the link is valid.
- Writes the triple $(l, 0, ts+1)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (l, v, ts))$.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, 0, ts+1))$.

Creating Links (Algorithm 2). For a new link, under cwrite_new(l), client does the following.

- Writes the triple $(l, 1, 0)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, 1, 0))$.
- Increments count $\mathcal{C}$ by one.

Reading Links (Algorithm 3). For cread(l), client actions involve the following.

Algorithm 3 cread(l)

```
1: Read (v, ts) for link l.
2: if v ≠ 1 then
3:     Abort
4: end if
5: Write (l, v, ts + 1)
6: H(R) = H(R) ∪ (l, v, ts)
7: H(W) = H(W) ∪ (l, v, ts + 1)
```

Algorithm 4 verify()

```
1: Let v ← C
2: while v ≠ 0 do
3:     Read (l, v, ts)
4:     H(R) = H(R) ∪ (l, v, ts)
5:     v = v − 1
6: end while
7: if H(R) ≠ H(W) then
8:     AbortAndReturn
9: end if
```

- Reads the tuple (v, ts) for link l from the provider.
- Checks that v is 1, that is the link is valid.
- Writes the triple $(l, v, ts + 1)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (l, v, ts))$.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, v, ts + 1))$.

Verification (Algorithm 4). After a sequence of operations, the client requests all triples from the server. For each triple received, the client updates $\mathcal{H}(\mathcal{R})$ and decrements $\mathcal{C}$. When $\mathcal{C} = 0$, client checks whether $\mathcal{H}(\mathcal{R}) = \mathcal{H}(\mathcal{W})$. The collision-resistant property of hash function $\mathcal{H}$ ensures that if the provider operates incorrectly, then $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ with high probability. We note that the client does not need to store the downloaded triples locally and can compute the hashes while streaming to compute $\mathcal{H}(\mathcal{R})$. As in memory checking (Section 3.1), longer operation sequences result in lower amortized per-operation verification cost.

4.3 QA for SQL Queries : An Overview

We use the BasicQA protocol for transaction-level QA as follows. For each query Q in a transaction, client submits Q to the provider and receives some results as a response. The client then checks the results for partial correctness (not completeness). For example, client checks whether all result tuples satisfy the query predicates. Once partial correctness is verified, using the query results, client constructs and records the hashes for all tuples that were read or written as part of query execution. At transaction commit, using the recorded tuple hashes, client initiates the BasicQA protocol. Section 4.4 details the derivation of BasicQA operation sequences from query results for SQL queries.

Note that in ConcurDB, a client does not wait to submit subsequent transactions to the provider. Execution of BasicQA operations for a transaction is initiated in a parallel client thread. Thus, transaction execution is not stalled by QA as is the case with online QA. In a multi-client scenario, each client performs the BasicQA operations in a parallel thread and continues to execute new transactions.

A client is required to store the tuple hashes only temporarily. As soon as the BasicQA operations are performed by the parallel thread for a particular transaction T, client discards the tuple hashes recorded for T. In practice, since the BasicQA operations involve far less computations than server-side transaction execution, the BasicQA operations for a transaction are expected to complete before a subsequent transaction commits.

Initiating the BasicQA protocol at transaction commit enables two optimizations. The first optimization is elimination of redundant operations. For example, if the same tuple was read twice in a transaction, with no updates in between, then we only perform the BasicQA operation once for the tuple link. The second optimization is reduction in client-server communication. BasicQA operations for a transaction are batched and performed via a single round trip message to the provider.

When verification for completeness and replay attack detection is desired, a client or the data owner initiates a special transaction that executes the final memory checking step wherein all triples are downloaded from the provider. The final step verifies completeness and checks replay attacks for all committed transactions.

There is a performance-security trade-off for the verification step. Delaying the verification step reduces the per-transaction QA cost. However, delaying the verifications also delays the detection of malicious behavior. In real-world deployments, the verification can be initiated at times of low transaction load e.g., every night. For amortized constant QA cost per transaction, verifications must be performed periodically after n/r transactions have committed. Here, n is the number of tuples initially uploaded by the owner; and r is the average number of tuples read and written per transaction.

4.4 Mapping SQL Queries to BasicQA Operations

Range (Select) Queries. Consider a range query for all tuples with keys in the range $[L, U]$, $L \leq U$. Let $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ denote the set of tuples in the query result. t_0 and t_{r+1} are two boundary tuples included in the query result to ensure completeness.

For completeness, t_0 must be the immediate predecessor of t_1 and t_{r+1} must be the immediate successor of tuple t_r. That is, t_0 and t_{r+1} are required to satisfy the following conditions:

- $t_0.key < L$ and $\nexists\, t_i$, such that $t_0.key < t_i.key < L$.
- $t_{r+1}.key > U$ and $\nexists\, t_i$, such that $t_{r+1}.key > t_i.key > U$.

At query execution, client checks the following for correctness of result set $\mathcal{R}$:

- $t_0.key < L$.
- $t_{r+1}.key > U$.

- $L \leq t_i.key \leq U$, where $1 \leq i \leq r$.

Additionally, client records the hashes of all tuples in $\mathcal{R}$. In the parallel QA step, using the recorded hashes, the client executes the following BasicQA operations sequence.

- cread(l); where $l = h(h(t_{i-1})||h(t_i))$, $1 \leq i \leq r+1$.

In order to violate correctness or completeness the provider has three choices. The first choice is to introduce a fake tuple in the result. The second choice is to omit a valid tuple from the result. The final choice is to resend a tuple that was previously updated or deleted by client. The first two choices would result in a BasicQA operation on a link that was neither written by the owner at database upload time nor written by the client using the $cwrite_new$ operation. The third choice would result in a BasicQA operation on a link that was previously made invalid by client using the $cwrite$ operation. As a result, in the final QA step, for all provider choices, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ will not be equal. Therefore, QA infringements are guaranteed to be detected by ConcurDB's offline memory checking-based QA.

Update Queries. Consider an update query that modifies all tuples with keys in the range $[L, U]$, $L \leq U$. The client first executes a select query for all tuples in the range $[L, U]$, $L \leq U$. Let $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ denote the set of tuples in the query result. t_0 and t_{r+1} are two boundary tuples included for completeness as in the case of range queries. Similar to the case of range queries, client performs checks for correctness on set $\mathcal{R}$.

The client then locally modifies the tuples to create the set $\mathcal{R}' = \{t_0, t'_1, t'_2, ..., t'_r, t_{r+1}\}$, where t'_i is the updated version of tuple t_i. Boundary tuples are not updated since they fall outside the query range.

After local modifications, client records the hashes of all tuples in sets $\mathcal{R}$ and $\mathcal{R}'$. Finally, client issues the update query to provider and the provider updates database tuples.

In the parallel QA step, a client constructs and executes a sequence of BasicQA operations using the stored tuple hashes as follows.

- cread(l); where $l = h(h(t_{i-1})||h(t_i))$, $1 \leq i \leq r+1$.
- cwrite(l); where $l = h(h(t_{i-1})||h(t_i))$, $1 \leq i \leq r+1$.
- cwrite_new(l); where $l = h(h(t'_{i-1})||h(t'_i))$, $1 \leq i \leq r+1$, $t'_0 = t_0$.

$cwrite$ operations invalidate old links and $cwrite_new$ operations add new links for the updated tuples.

Insert Queries. Consider an insert query to add a tuple t with key k. The client first executes a select query for the range $[k, k]$. Let $\mathcal{R} = \{t_l, t_u\}$ denote the set of tuples in the select query result. t_l and t_u are required to be the immediate predecessor and successor, respectively, of tuple t. The client then records $h(t_l)$, $h(t_u)$, and $h(t)$. Finally, the client submits an insert query to the provider. The provider adds the new tuple to the database.

BasicQA operations for the insert query are as follows:

- cread(l); where $l = h(h(t_l)||h(t_u))$.
- cwrite(l); where $l = h(h(t_l)||h(t_u))$.
- cwrite_new(l); where $l = h(h(l)||h(t))$.
- cwrite_new(l); where $l = h(h(t)||h(u))$.

The old link between t_l and t_u is invalidated and two new links are added for tuple t. For clarity, we illustrated an insert with a single tuple. Extensions to insert-select queries are straight-forward.

Delete Queries. Delete queries are processed similar to updates queries – the client first executes a select query to fetch all tuples that would be deleted. The client then verifies correctness, records hashes, and submits the delete query to provider.

If $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ is the select query result, then the BasicQA operation sequence for the delete query with range $[L, U]$ ($L \leq U$) is the following:

- cread(l); where $l = h(h(t_{i-1})||h(t_i))$, $1 \leq i \leq r+1$.
- cwrite(l); where $l = h(h(t_{i-1})||h(t_i))$, $1 \leq i \leq r+1$.
- cwrite_new(l); where $l = h(h(t_0)||h(t_{r+1}))$.

Join Queries. We provide two mechanisms for join processing – client-side join and nested join. To illustrate the two mechanisms, consider a join query:

$$\sigma_{R.a=S.b \ and \ R.a \in [L_1, U_1] \ and \ S.b \in [L_2, U_2]}$$

where R and S are two relations; and a and b are search attributes of R and S, respectively.

If selectivity of both attributes is high, joins are processed client-side. The client executes and verifies two select queries $\sigma_{R.a \in [L_1, U_1]}$ and $\sigma_{S.b \in [L_2, U_2]}$. Using the query results, the client performs the join locally.

In the nested join mechanism, the relation with higher selectivity of the search attribute is first selected. In the example query suppose that $R.a$ has higher selectivity. The client first executes the range query $\sigma_{R.a \in [L_1, U_1]}$, performing checks for correctness and gathering tuple hashes. For each tuple t in the query result, client executes and verifies a select query $\sigma_{t.a=S.b}$.

In case of low selectivity of search attributes join efficiency in ConcurDB will be lower than tree-based approaches. Tree-based approaches are more efficient for join processing and thus more suitable for read-intensive OLAP applications [2]. Our focus is OLTP applications with frequent data updates. Hence, we design ConcurDB for efficient multi-client update scenarios.

5 ConcurDB: Achieving Concurrency

5.1 Multi-Client Offline QA

The security of ConcurDB's offline QA scheme relies on two hash values $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ (Section 4.2). Initially, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ are computed by the data owner at database upload time. In a single client scenario, the owner transfers the initial hashes and a count $\mathcal{C}$ to the only client. We collectively refer to the two hashes and count as the *authentication set* $\mathcal{A} = \{\mathcal{H}(\mathcal{R}), \mathcal{H}(\mathcal{W}), \mathcal{C}\}$. As part of the QA step for each transaction, client updates the elements of $\mathcal{A}$. The authentication set $\mathcal{A}$ remains with the single client until the final QA step.

In a multi-client scenario, transactions are executed concurrently by many clients. Each client performs the

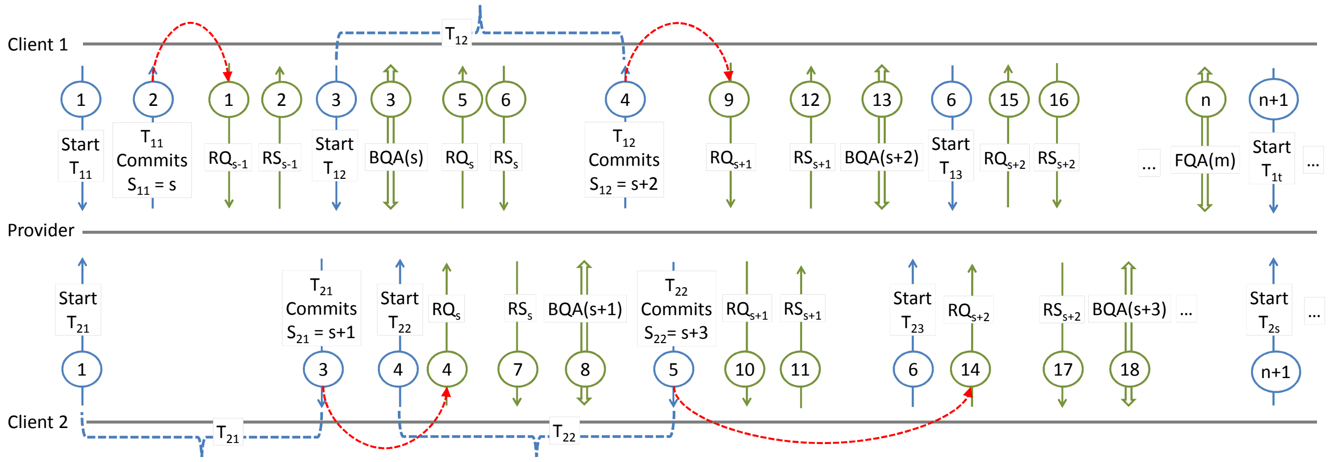

Fig. 1: Sample execution of ConcurDB's offline QA mechanism for two concurrent clients. Blue steps represent transaction execution. Green steps represent offline QA operations. Steps with the same sequence number and color, or with different colors occur in parallel except for the dependency indicated by red arrows. Dotted blue lines indicate the interval during which queries of a transaction execute. T_{ij} denotes j^{th} transaction of client i. S_{ij} denotes sequence number of transaction T_{ij}. $RQ_s = E_k(N, s, C_i)$ is the request for authentication set of transaction with sequence number s (Section 5.2). $RS_s = E_k(N, s, C_i, C_j, \mathcal{A})$ is the response to request for authentication set of transaction with sequence number s. $BQA(s)$ represents a BasicQA step for transaction with sequence number s (Section 4.2). $FQA(s)$ denotes the final QA step for a batch of m transactions (Section 4.2).

BasicQA operations for its transactions in a parallel thread. Since BasicQA involves updates to the set $\mathcal{A}$, all clients need access to $\mathcal{A}$. However, in order to ensure QA, only one copy of $\mathcal{A}$ can exist[4] and clients need to access the single copy. This is analogous to the root hash in tree-based ADS. However, unlike online tree-based QA transaction execution is not stalled by QA.

This requires *serializing* the BasicQA step while executing the queries in parallel. In fact, for consistency, databases operate under a serializable mode. This ensures that a concurrent transaction schedule is equivalent to a serial schedule of transactions [13]. Therefore, to ensure QA consistency, the order of BasicQA steps must match a serial order of committed transactions.

This is aided by exchanging the *authentication set* which includes the read and write hashes and additional information for synchronization. In the following, we discuss the exchange protocol using the untrusted provider as a communication hub.

5.2 Authentication Set Exchange Protocol

The idea behind the protocol is that the provider assigns a unique incrementing sequence number for each query. In effect, the provider commits to a serial transaction schedule. The BasicQA operations are then performed in the order of assigned sequence numbers. The protocol design ensures that – i) the provider cannot skip sequence numbers (i.e., cannot skip a transaction execution altogether), ii) cannot reuse old sequence numbers (i.e, cannot perform a replay attack using old authentication set), and iii) cannot tamper with client messages.

Setup. All clients and the data owner share a key K which can be shared using broadcast encryption [4]. We denote by $E_K(M)$, the encryption of message M with key K. The initial database upload by the owner is considered as a transaction with sequence number 0. The server assigns sequence numbers to client transactions starting from 1.

Protocol Intuition. As part of the exchange protocol, each client performs two essential operations.

- Before commencing the BasicQA operations for a transaction with sequence number s, a client acquires the authentication set (from another client using the server as the communication conduit) resulting from BasicQA for transaction with sequence number $s-1$.
- After completing the BasicQA operations for a transaction with sequence number s, a client responds to a request (from another client) for authentication set with sequence s.

Exchange Protocol (Figure 1). Formally, the protocol has two operations:

- AcquireSet (Algorithm 5): The client performing the query that is assigned sequence number s acquires the updated authentication set from the client that executed the query assigned sequence number $s-1$. This includes the following steps:
  - Generate a random nonce r.
  - Push the message $E_k(r, s-1, \mathcal{C})$ to the provider, where $\mathcal{C}$ is the unique client identifier.
  - Wait for the response message $E_k(r', s', \mathcal{C}', \mathcal{C}'', \mathcal{A})$ Here, C'' is the responding client's identifier and $\mathcal{A}$ is the authentication set for sequence s'.
  - Decrypts the response message and verifies that $r' = r$, $s' = s-1$, and $\mathcal{C}' = \mathcal{C}$.
- UploadSet (Algorithm 6): The client executing the query with sequence number s waits for request

4. For partitioned databases, one copy can exist for each partition.

Algorithm 5 AcquireSet

```
1:  r ← random nonce
2:  s ← Query Sequence Number
3:  C ← Client Identifier
4:  Push E_k(r, s, C) to the server
    // Wait for response message
5:  R ← E_k(r', s', C', C'', A)
6:  if r' ≠ r then // Nonce does not match request
7:      AbortAndReturn
8:  end if
9:  if C' ≠ C then // Wrong client ID in response
10:     AbortAndReturn
11: end if
12: if s' ≠ s − 1 then // Incorrect serialization
13:     AbortAndReturn
14: end if
15: return R
```

Algorithm 6 UploadSet

```
1:  s ← Query Sequence Number
2:  C ← Client Identifier
    // Wait for request message
3:  R ← E_k(r, s', C')
4:  if s' ≠ s + 1 then // Incorrect serialization
5:      AbortAndReturn
6:  end if
7:  if C has responded for s + 1 then // Replay attack
8:      AbortAndReturn
9:  end if
10: Push E_k(r, s, C', C, A) to the server
```

message issued by the client with sequence number $s+1$ (using AcquireSet) and upon receiving the request performs the following steps:

- Wait for a request message with sequence number $s+1$. The request message is of the form $E_k(r, s+1, \mathcal{C}^{'})$, where $\mathcal{C}^{'}$ is the requesting client's identifier.
- Decrypt the request message.
- Ensure that the client has not previously responded for the sequence $s+1$. The client stores locally the sequence number for the last query it responded to.
- Push the message $E_k(r, s, \mathcal{C}^{'}, \mathcal{C}^{''}, \mathcal{A})$ to the server.

Figure 1 illustrates our concurrent offline QA mechanism for two clients.

5.3 Protocol Analysis

Correctness. When serialized correctly and sequence numbers are assigned to queries in the order of commits, the *authentication set* is also updated following the order of query commits. This ensures consistency of the authentication set.

Security Against Malicious Server. Since the non-interacting clients use the server as a conduit for relaying messages, an actively malicious server can mount replay attacks by exposing older versions of the *authentication set* to the clients. In the absence of direct inter-client communication, the strongest form of consistency that can be achieved is *fork consistency* [17] – where the server selectively replays the states of server-side data structures and presents different (possibly conflicting) views of the system to different clients. Importantly, these views cannot be consistently unified later leading to immediate detection of the forking.

In ConcurDB, fork consistency is achieved by including a randomly generated nonce in each exchange protocol message.

Theorem 1. *In ConcurDB, if the malicious server forks client views by replaying messages, then these conflicting views cannot be unified later without being detected.*

Proof. W.l.o.g. consider a client $\mathcal{C}_1$ that is assigned query sequence number s by the malicious server $\mathcal{S}$. Let $\mathcal{C}_2$ be the client next in sequence with sequence number $s+1$. Once $\mathcal{C}_2$ completes its query, it requests the updated authentication set from $\mathcal{C}_1$ by pushing message $M = E_k(r, s+1, \mathcal{C}_2)$ to the server, where r is a randomly generated nonce of appropriate size.[5]

Consider that the server now mounts a replay attack by assigning sequence number $s+1$ to another client $\mathcal{C}_3$. Then, $\mathcal{C}_3$ will push request message $M^{'} = E_k(r^{'}, s+1, \mathcal{C}_3)$ for acquiring the authentication set.

To prevent detection, $\mathcal{S}$ must present either M or $M^{'}$ to $\mathcal{C}_1$, *but not both*, since clients verify whether they have already responded for a particular sequence number (Algorithm 6, steps 7 - 9).

Let, $\mathcal{C}_1$ respond with $R^{'} = E_k(r^{'}, s, \mathcal{C}_2, \mathcal{C}_1, \mathcal{A})$ to $M^{'}$. $\mathcal{C}_2$ will not accept $R^{'}$ as a reply for M since it first verifies that the included nonce in the message $r^{'} \neq r$ (Algorithm 5, steps 6 - 8). Also, $\mathcal{S}$ cannot forge the response message for M without the encryption key K.

As a result, $\mathcal{C}_2$ and $\mathcal{C}_3$ will have different views of the database and the authentication set henceforth. To unify these views later on, $\mathcal{S}$ will need to recompute the hashes in the authentication set, as well as forge a response message for a request issued by a particular client $\mathcal{C}_x$. This is not possible without access to K. □

5.4 Detection of Faulty Transactions

ConcurDB's offline QA mechanism described so far only detects QA infringements for a batch of transactions. Specific transactions for which the provider cheated or malfunctioned are not identified. However, in certain applications, identification of faulty transactions may be necessary for corrections.

We extend ConcurDB for detection of faulty transactions as follows. During offline QA, the provider is

5. The size of the nonce can be set according to the security parameter in order to achieve negligible likelihood of collisions among values generated by multiple clients.

required to record the sequence of all BasicQA operations. The provider-recorder operations sequence is authenticated by adding a new hash $\mathbb{H}(\mathcal{O})$ to the clients'-side authentication set. $\mathbb{H}(\mathcal{O})$ records (in sequence) all BasicQA operations performed by clients.

Once offline QA detects a violation, that is, $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ in the final QA step (Section 4.2), a switch over to online QA occurs. The online QA mode repeats verification of all BasicQA operations. By design, online QA identifies the faulty operation immediately.

Note that the switch over to online QA occurs only if a violation is detected by offline QA. The switch over condition is important for efficiency. Recall from Section 1 that online QA has a complexity of $O(\log n)$ per operation as compared to $O(1)$ complexity for offline QA. For providers that malfunction or cheat occasionally the switch over condition ensures lower offline QA costs for most of the time. The higher price for online QA applies only when an occasional fault occurs. We emphasize that the assumption for occasional violations is reasonable in practice. If a provider malfunctions or cheats often, then switching providers may be a more prudent decision than using a more expensive online QA solution considering that online QA would also limit transaction throughputs.

The above augmentations add only a constant number of operations for clients and provider. Hence, the amortized constant cost of offline QA is preserved. Further, clients can continue to execute new transactions.

6 Experiments

Implementation and Setup. ConcurDB implementation is split into client-side and server-side libraries. The client-side Java library transparently performs all QA related operations, including query rewriting; and BasicQA and concurrency protocols. The Server-side library comprises a set of stored procedures that implement provider-side QA functionality. For the authentication set exchange protocol (Section 5.2), we use the RabbitMQ message broker. The total LOC is ≈11K.

We use a test bed of three servers. Each server has 2 Intel Xeon quad-core CPUs at 3.16GHz and 8GB RAM. One server hosts the MySQL database (v 5.6.17) and the RabbitMQ message broker playing the role of service provider. Remaining two servers host clients[6].

Comparison with Previous Work: We compared ConcurDB with Jain et al. 's MBT implementation. To the best of our knowledge, this is the only prior work on query authentication for dynamic outsourced databases in multi-client settings that does not rely on the data owner to perform all updates, trusted hardware and complex cryptographic primitives.

While there are solutions that support verifiable concurrent queries for key-value stores in multi-client settings e.g., [1, 30], the techniques presented therein cannot be trivially extended to support more expressive SQL queries. Also, these solutions rely on server-side trusted hardware support which is often an unrealistic assumption for practical settings.

On the other hand, query authentication techniques for SQL queries that are based on cryptographic primitives (and can be applied to multi-client settings) e.g., [32, 33] are slow due to large proofs (poly-logarithmic in the size of the query) that must be transferred over the network. Also, these solutions rely on server-side code running in a trusted environment to perform complex cryptographic operations.

MBT Implementation. We implemented MBT similar to Jain et al. 's implementation on top of MySQL [12]. We do not endow the MBT implementation with replay attack detection. Permitting replay attacks on MBT enables a comparison of concurrency-related characteristics only. The ConcurDB approach on the other hand, performs the full offline QA protocol. Hence, we note that in real deployments with replay attack detection, MBT will be slower than the performance reported in our results.

Query Throughput. We compare ConcurDB and MBT for both select and update queries. The data set consists of a relation with 10 million random integer keys. Each client transaction performs a select or update query for a range of tuples. A total of 10K transactions are executed in each test.

Figures 2 and 3 shows how overall query throughput scales for ConcurDB and MBT for selects and updates with increasing number of clients in the system, respectively. Overall, ConcurDB performs 1.5x - 4x better than the MBT approach with no replay attack detection. Adding replay attack detection to MBT will widen the performance gap further. The results directly stem from the $O(\log n)$ complexity of MBT versus the $O(1)$ amortized execution and data transfer complexity of ConcurDB's offline mechanism.

As discussed in Section 4, the procedure for performing deletions in ConcurDB is similar to the procedure for performing updates (Section 4.4). Therefore, deletion query throughput is expected to follow the same trend as the query throughput for updates (Figure 3). These results are omitted due to space constraints

Data Transfer. Figures 4(a) and 4(b) report the total network-level data transferred. Since we keep the total number of transactions constant (10K) in each test, data transfer is similar for 1,2,4,8,16, or 32 concurrent clients. Hence, in Figures 4(a) and 4(b) we report the average data transfer with error bars indicating the minimum and maximum.

For small result set sizes, ConcurDB exhibits up to 4x reduction in data transfer as compared to MBT. For large result set sizes, the data transfer gap between ConcurDB and MBT is lower. This results from the large fanout of MBT. Large result sets include tuples from multiple leaf nodes. Since multiple leaf nodes share common parent nodes at upper levels, the per tuple *verification object* overhead reduces. ConcurDB's data transfer on the other

6. We simulate clients using the BenchmarkSQL tool.

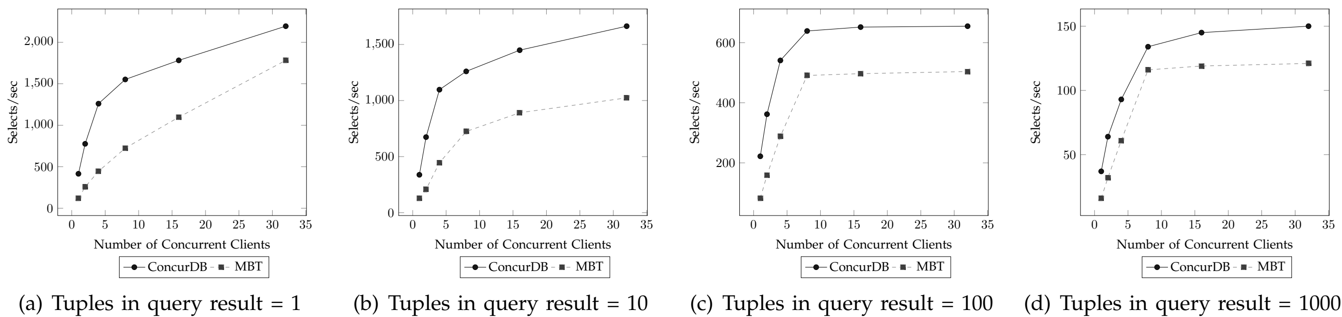

Fig. 2: Select queries executed per second vs. number of concurrent clients (Higher is better). Overall query throughput for ConcurDB scales better than MBT with increasing clients. ConcurDB is up to 4x faster than MBT.

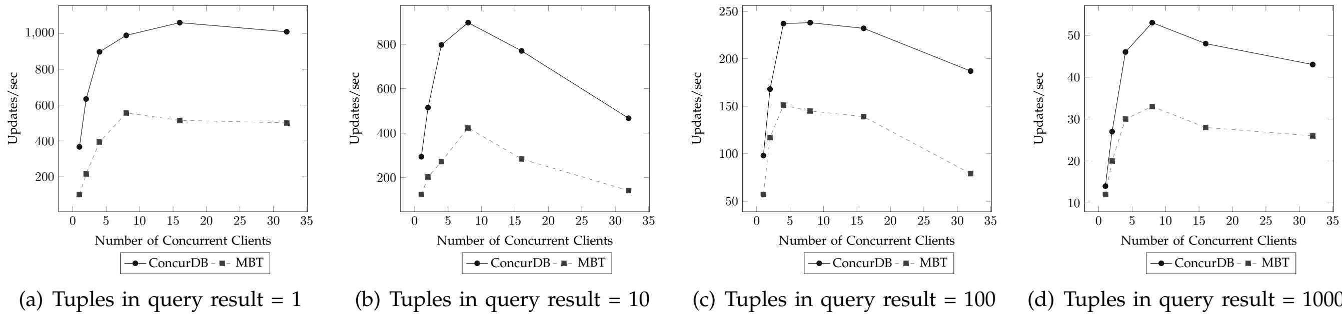

Fig. 3: Update queries executed per second vs. number of concurrent clients (Higher is better). Overall query throughput for ConcurDB scales better than MBT with increasing clients. ConcurDB is up to 3x faster than MBT.

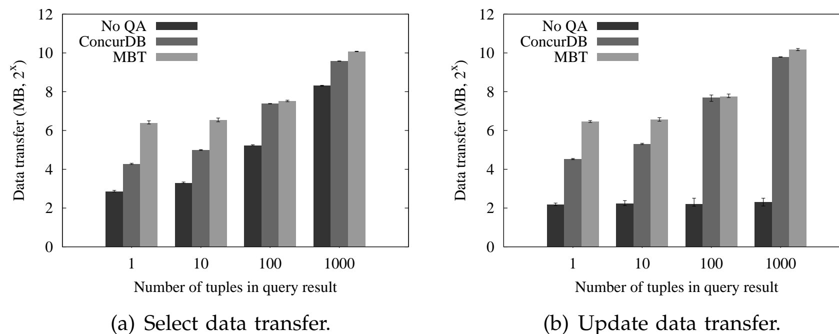

Fig. 4: Data transfer comparison of ConcurDB and MBT [12] with varying result set size (1,10,100,1000). ConcurDB exhibits up to 4x reduction in data transfer as compared to MBT.

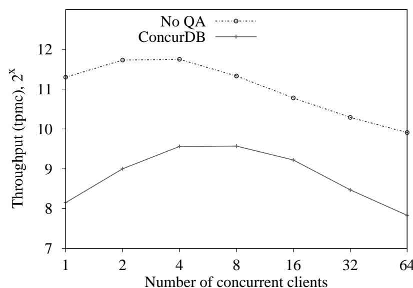

Fig. 5: Comparison of ConcurDB and MySQL (No QA).

hand increases linearly with query result size.

In a database with no QA, updates do not return results to clients except for an acknowledgement. Hence, data transfer is nearly constant irrespective of result size (Figure 4(b)). For QA solutions, updates involve search operations to identify the correct set of target tuples. Hence, in QA solutions, data transfer for updates is not constant but increases with result set size.

TPC-C. We use a TPC-C scale factor of 100. TPC-C transactions' distributions were set as per the specification[7]. A total of 100K transactions were executed for each test. As per the TPC-C benchmark, we measure throughputs as the number of new order transactions executed per minute (tpmc).

Figure 5 shows the throughput results in comparison with a plain MySQL database with no QA. Throughputs are plotted on the logarithmic scale. Even for complex TPC-C transactions, ConcurDB maintains a QA overhead similar to the case of simple select and update queries.

7. 45% new order, 43% payment, 4% order status, 4% delivery, and 4% stock level.

7 Conclusions

ConcurDB is an offline, concurrent, query authentication (QA) scheme that supports updates by multiple clients. ConcurDB eliminates bottlenecks on updates; increases transaction concurrency by decoupling transaction execution and verification; and detects replay attacks efficiently. For updates, ConcurDB performs up to 4x better that tree-based QA. Using the TPC-C benchmark, we also demonstrate that ConcurDB can achieve efficient QA for full-fledged OLTP applications.

References

[1] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064030

[2] S. Bajaj and R. Sion, "Correctdb: Sql engine with practical query authentication," *VLDB*, vol. 6, no. 7, pp. 529–540, 2013. [Online]. Available: http://dx.doi.org/10.14778/2536349.2536353

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," ser. SFCS, 1991. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1991.185352

[4] D. Boneh, C. Gentry, and B. Waters, "Collusion resistant broadcast encryption with short ciphertexts and private keys," in *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, ser. CRYPTO'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 258–275. [Online]. Available: http://dx.doi.org/10.1007/11535218_16

[5] Y. Chen and R. Sion, "To cloud or not to cloud?: Musings on costs and viability," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 29:1–29:7. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038945

[6] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Advances in Cryptology - ASIACRYPT 2003*, C.-S. Laih, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 188–207.

[7] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic data publication over the internet," in *Journal of Computer Security*, 2003, pp. 291 – 314.

[8] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan, "How efficient can memory checking be," Tech. Rep., 2008.

[9] B.-M. Goi, M. U. Siddiqi, and H.-T. Chuah, "Computational complexity and implementation aspects of the incremental hash function," *IEEE Trans. on Consum. Electron.'03*. [Online]. Available: http://dx.doi.org/10.1109/TCE.2003.1261226

[10] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *DISCEX*, 2001.

[11] H. Hu, Q. Chen, and J. Xu, "Verdict: Privacy-preserving authentication of range queries in location-based services." in *ICDE*, pp. 1312–1315. [Online]. Available: http://dblp.uni-trier.de/db/conf/icde/icde2013.html#HuCX13

[12] R. Jain and S. Prabhakar, "Trustworthy data from untrusted databases," *2014 IEEE 30th International Conference on Data Engineering*, vol. 0, pp. 529–540, 2013.

[13] M. Kifer, A. Bernstein, and P. Lewis, *Database Systems: An Application-oriented Approach.* Pearson/Addison-Wesley, 2006, no. v. 2. [Online]. Available: http://books.google.com/books?id=9W0_AQAAIAAJ

[14] J. Krupp, D. Schröder, M. Simkin, D. Fiore, G. Ateniese, and S. Nuernberger, "Nearly optimal verifiable data streaming," in *Proceedings, Part I, of the 19th IACR International Conference on Public-Key Cryptography — PKC 2016 - Volume 9614*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 417–445. [Online]. Available: https://doi.org/10.1007/978-3-662-49384-7_16

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *SIGMOD'06*.

[16] ——, "Authenticated index structures for aggregation queries," vol. 13, no. 4. New York, NY, USA: ACM, Dec. 2010, pp. 32:1–32:35. [Online]. Available: http://doi.acm.org/10.1145/1880022.1880026

[17] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *OSDI*, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251263

[18] X. Lin, J. Xu, and H. Hu, "Authentication of location-based skyline queries," in *CIKM'11*. ACM.

[19] R. C. Merkle, "A certified digital signature," in *Advances in cryptology*, 1989, pp. 218–238.

[20] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM TOS, Volume 2 Issue 2*, pp. 107 – 138, 2006.

[21] M. Narasimha and G. Tsudik, "Dsac: integrity for outsourced databases with signature aggregation and chaining," in *CIKM*. ACM, 2005.

[22] H. Pang and K.-L. Tan, *Query Answer Authentication*, ser. Synthesis Lectures on Data Management, 2012. [Online]. Available: http://dx.doi.org/10.2200/S00405ED1V01Y201202DTM024

[23] H. Pang, J. Zhang, and K. Mouratidi, "Scalable verification for outsourced dynamic databases," *VLDB'09*.

[24] H. Pang, A. Jain, K. Ramamritham, and K. Tan, "Verifying completeness of relational query results in data publishing," in *SIGMOD*, 2005, pp. 407 – 418.

[25] S. Papadopoulos, D. Papadias, W. Cheng, and K.-L. Tan, "Separating authentication from query execution in outsourced databases." in *ICDE*, 2009, pp. 1148–1151. [Online]. Available: http://dblp.uni-trier.de/db/conf/icde/icde2009.html#PapadopoulosPCT09

[26] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Optimal verification of operations on dynamic sets," ser. CRYPTO'11, Berlin, Heidelberg, 2011, pp. 91–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033036.2033045

[27] R. C. W. Phan and D. Wagner, "Security considerations for incremental hash functions based on pair block chaining," *Comput. Secur. '06, 131-136*. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2005.12.006

[28] Y. Qian, Y. Zhang, X. Chen, and C. Papamanthou, "Streaming authenticated data structures: Abstraction and implementation," in *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, ser. CCSW '14. New York, NY, USA: ACM, 2014, pp. 129–139. [Online]. Available: http://doi.acm.org/10.1145/2664168.2664177

[29] S. Singh and S. Prabhakar, "Ensuring correctness over untrusted private database," in *EDBT'08*. ACM.

[30] R. Sinha and M. Christodorescu, "Veritasdb: High throughput key-value store with integrity," *IACR Cryptology ePrint Archive*, vol. 2018, p. 251, 2018.

[31] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *SIGMOD*. ACM, 2009, pp. 5 – 18.

[32] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vsql: Verifying arbitrary sql queries over dynamic outsourced databases," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 863–880.

[33] Y. Zhang, J. Katz, and C. Papamanthou, "Integridb: Verifiable sql for outsourced databases," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1480–1491. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813711

[34] Y. Zhou and C. Wang, "An online query authentication system for outsourced databases." in *TrustCom'13*. [Online]. Available: http://dblp.uni-trier.de/db/conf/trustcom/trustcom2013.html#ZhouW13

Sumeet Bajaj is the head software architect at Private Machines Inc. He received his PhD from Stony Brook University in 2014. His research interests include Network Security, Databases, Distributed and Concurrent Systems. His thesis research involved designing and building data management systems for regulatory compliance. His industry experience involves building cloud services, financial trading, and ERP systems.

Anrin Chakraborti is a PhD student at Stony Brook University and part of the Network Security and Applied (NSAC) Lab. His research interests include privacy preserving mechanisms, cloud security and applied cryptography.

Radu Sion is a Professor at Stony Brook University, the Director of the National Security Institute, and the CEO of Private Machines Inc. Radu's research is in Cyber Security and Large Scale Computing. He has published 85+ peer reviewed works in top venues, and has organized 65+ conferences. Radu has received the National Science Foundation CAREER award for his work on cloud computing security. Radu has worked with and received funding from numerous industry and government partners, including the US Air Force, the Office of the Secretary of Defense, the Department of Homeland Security, the US Army, the Inteligence ARPA, the Office of Naval Research, Northrop Grumman, IBM, NOKIA, Motorola, Xerox Parc, Microsoft, SAP, CA Technologies, the National Science Foundation, and many others.